Using Culvert to Manage Component Subversions

Julius Davies†
† Department of Computer Science, University of British Columbia, Canada
juliusd@cs.ubc.ca

ABSTRACT

A software system often includes a set of library dependencies and other software artifacts necessary for the system's proper operation. Reuse of components greatly improves developer productivity, but reuse also entails long-term maintenance costs. As we showed in our previous report, components can suddenly suffer from legal problems, security holes, and other problems long after their initial integration and deployment into a successful running system. Existing research in component based software engineering and software reuse rarely acknowledges this tradeoff.

In our submission to the ICSE 2011 Student Research Competition we performed a manual case study of components in a proprietary eCommerce web application to validate our approach. In this followup work we have developed and deployed an automated tool (`http://mavian.org/culvert/`) that translates Debian and Ubuntu package dependencies into a format compatible with the Maven2 build system. Developers can use this tool to significantly minimize costs of component reuse over the lifetimes of their software projects.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable libraries

General Terms

Management, Legal Aspects, Security

Keywords

Reuse, licensing, security, maintenance

1. PROBLEM AND MOTIVATION

Software engineers build much of the world's running software by combining independently developed components. This 'Lego' approach to software development, popularized in Douglas Coupland's novel *Microserfs* [1], is increasingly important to all sectors of the world economy. Most organizations cannot afford to write their own operating system or database engine from scratch. Even small reusable software libraries (e.g., MP3 decoding, photo viewing, secure encryption) are often adequate to fulfill the end-user's needs. Re-implementing libraries such as these from scratch would require time and effort, and important details required for proper operation could easily be overlooked. But the Lego analogy for software is misleading. It suggests a style of component based software engineering (CBSE) that, with the placement of the final Lego, results in a finished product. Existing research shows that as long as a software project has active users and developers, the software will undergo continuous change, much like a living organism [2]. Each Lego, so to speak, is an independent body in motion. Existing research on CBSE rarely acknowledges this dynamic, e.g., Lau & Wang's survey [3] assumes each component is a finished work.

Our own research shows that systems occasionally *must* upgrade (or downgrade) the components they consume to different versions. The recent Diginotar security crisis is an extreme example: Chrome, Firefox, Internet Explorer, Citrix, Microsoft Windows, Mac OS X, Redhat Enterprise Linux, etc. — any component with any sort of security subsystem required immediate, urgent upgrade. The chance for mandatory upgrades significantly perturbs the presumed cost-benefit of CBSE's Lego bargain. Meanwhile, the upgrade itself can also cause problems. For example, consumers of the widely used Java logging library, Log4j, break their own systems should they try version 1.3. Only version 1.2 works correctly.

2. BACKGROUND AND RELATED WORK

> *The most promising attack continues to be re-use, and re-use of COTS programs in particular. [...] It remains true that "The most radical possible solution for constructing software is not to construct it at all. [4]"*
>
> (Frederick P. Brooks Jr., OOPSLA-2007 Panel [5])

> *The open-source community has made an astonishing array of software available which I can use by just typing '`apt-get install`'.*
>
> (Grady Booch [5])

Class Libraries and Frameworks present an increasingly complex and constantly changing sea of APIs that must be mastered to create even the simplest of applications.
(Dave Thomas [5])

In 2011 we are in the midst of yet another software productivity miracle. In less than an hour, a single developer can download and integrate into their own project a software ecosystem that would require tens of thousands of years if the developer wrote it all herself [6]. Within the first hour of writing a new system from scratch, an individual developer's new software system may already include the following:

1. A complete operating system with various file systems and networking stacks.
2. A programming platform (such as Java or .NET) with extensive libraries and automatic garbage collection.
3. A web server, an application server, and mechanisms for responding to inbound web service calls.
4. A transactional relational database management system (RDBMS).
5. Outbound web service connections to credit card payment processing or other 3rd party services.

But these improvements do not come for free. Reusability has its own costs and risks. Initial costs are well documented in current research. Among these we see artifact price, integration difficulty [7], and the search costs required of developers to find and evaluate potential reusable artifacts [8, 9]. Long-term costs and risks from reusability include legal risk, security risk, future ease-of-maintenance, lack of widespread industry knowledge concerning the artifact (e.g., can we hire developers that know how to deal with the artifact?), and potential artifact end-of-life.

The object-oriented revolution of the 1990's gave rise to a strong research tradition in distributed and component-based software. The Microsoft library, DCOM, first introduced in 1996, stands for "Distributed Component Object Model." But the bulk of this tradition focuses on developers making *new connections* between components. Rarely does previous research touch on development activities and challenges that occur after the components are successfully connected. A recent survey on models of component-based software engineering (CBSE) [3] unintentionally highlights the gap we are addressing with its definition of an idealized component life cycle:

> First, components should be preexisting reusable software units [...]. Second, components should be produced and used by independent parties. [...] Third, it should be possible to copy and instantiate components [...]. Fourth, components should be composable into [...] even larger [...] composite components.

We believe a component life cycle should also include phases for component end-of-life and other long-term maintenance activities. For example, developers should be able to assess integrated components for version-specific vulnerabilities.

3. APPROACH AND UNIQUENESS

Reusable software libraries greatly improve productivity and software quality for the initial development of new software systems and features, since much of the necessary logic and functions can be delegated to library dependencies. But these libraries also create a critical dilemma for developers tasked with the ongoing future maintenance and evolution of the newly developed system: when should the dependencies be upgraded?

On the one hand, library upgrades can improve security, performance, and quality of the consuming system, but on the other hand, library upgrades can introduce new bugs, new instabilities, and in some cases newer library versions are not reverse-compatible with their previous incarnations, and they can "break the build," causing additional work for the developer conducting the upgrade.

In this paper we look at two ways Java developers have traditionally grappled with this dilemma. One approach, based on the Maven build system, generally avoids upgrades in favour of 'build reproducibility' and system stability. In this tradition, individual projects must manage their own dependencies, and Maven maintains a comprehensive set of library versions for projects to choose from. Another approach, based on the Debian GNU/Linux project, delegates the dilemma to a group of specialized volunteers who continuously test potential dependency upgrades, and in turn publish a series of dependency graph "editions" for the larger community's consumption.

To help us study these two approaches to dependency management we have developed a tool, Culvert, which allows developers of Maven-based systems to make use of Debian's dependency graph. Instead of choosing each specific version of each library their system uses, as typically required of Maven build scripts, users of Culvert simply choose one of the four Debian dependency graph editions: *unstable*, *testing*, *stable*, or *old-stable*. These four choices allow the developer to choose a degree of newness at a cost of confidence. For example, *unstable* contains newer libraries, but these libraries have undergone less testing, and so the Debian volunteers have less confidence in their suitability for production systems. On the other extreme of the range, *old-stable* contains older libraries, but these have undergone a more substantial degree of testing, including at least one year of production deployment on Debian systems.

To validate our tool's feasibility, we have conducted a case study using the same 81 Jar files we studied in our initial 2011 ICSE SRC report [10]. These 81 Jar files are configured "pinned" dependencies in the eCommerce system's Maven build scripts. We use the term 'version-pinning' here to refer to situations where the build system is configured to only ever download a single specific version of a library to satisfy a dependency, e.g., "The system can only use version 3.2.2 of this library." Debian rarely *pins* dependencies, since pinning breaks Debian's security-update mechanism. Maven build scripts predominantly *pin* dependencies to ensure consistent, reproduceable builds.

For our case study we have examined which versions of these 81 libraries Culvert would choose, and compared these to the versions already specified by the eCommerce developers. This gives us some idea of Culvert's feasibility, as well as an idea of the difficulty developers might face should they try to port their systems from Maven's "pinning" tradition to Culvert's more dynamic notion of dependency resolution.

Figure 1: A screenshot of Culvert's home page, accessed on Sunday April 15th, 2012.

3.1 Implementing Culvert

The goal of Culvert is to allow developers of Maven build scripts to download Debian Jar files. And thus the implementation for Culvert is a simple translation: for any given Debian package, obtain the necessary information, unpack it, and then offer it up to Maven over the HTTP protocol as required. Since Maven will issue HTTP requests based on values in its XML files (in particular, the `<repository>` and `<groupId>` elements), we have some leeway with how we place the Debian Jar files on our Culvert server. By organizing directories in particular arrangements we can expose additional Debian package information which may be useful to developers. In the scheme we developed for Culvert we expose the following information:

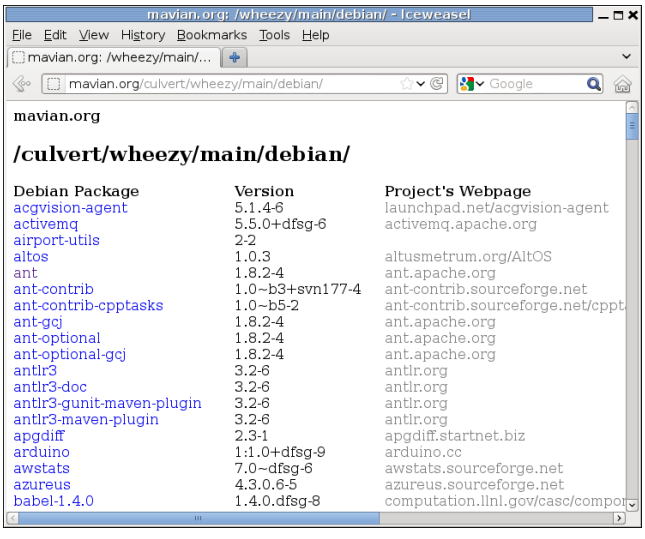

Figure 2: Browsing the repository: here Culvert lists all of Debian's "main" packages that contain at least one Jar file. The browsing feature also displays version information and project URLs extracted from Debian's packaging database.

Edition. As we mentioned earlier, Debian maintains four main editions of its version-dependency graph: *unstable*, *testing*, *stable*, or *old-stable*, although in truth there are also additional specialized editions (e.g., *experimental* and *security-updates*), but for our purposes we consider only these four primary editions. These editions are ranked in order from newest to oldest, with an commensurate ranking of lowest confidence to highest confidence, i.e., the edition containing newer library versions, *unstable*, provides the lowest confidence of stability and quality, whereas the edition containing older library versions, *old-stable*, provides the highest confidence. However, this improved confidence comes at a cost of increased age: the versions in *old-stable* will not contain the latest frameworks, functions, and utilities that might be needed for the integrating developer's project. These four editions represent the current active work of the Debian community. Debian also offers permanent names (e.g., *squeeze*) that transition through these stages until they are finally retired into the archive. We expose Debian's *edition* to Maven via the `<repository>` element:

<repository><url>
`http://mavian.org/culvert/squeeze/main/`
</url></repository>

Copyright Regime. Debian maintains three copyright regimes. Each Debian package must belong to one (and only one) of the three copyright regimes: *main contrib*, *non-free*. A large majority of packages (over 90%) belong to the *main* regime: these are packages that are compatible with the Debian Free Software Guidelines (DFSG), and, in addition, do not depend on any packages outside of *main*. The *contrib* regime contains the smallest number of packages: these are also compatible with the DFSG, but they have dependencies outside of *main*. Finally, the *non-free* packages generally are not compatible with the DFSG, but are made available to Debian community as a convenience with a strong *caveat emptor*, as they may contain patent issues, or other legal risks. We expose Debian's *copyright regime* to Maven via the `<repository>` element:

<repository><url>
`http://mavian.org/culvert/squeeze/main/`
</url></repository>

Figure 3: Here is how Culvert publishes the Jar files for *ant*, the popular Java build tool. The *.POM files, needed by Maven, are generated dynamically based on information found in Debian's package files, and include Debian's inter-package dependency information translated into a Maven2 compatible format.

```
<repositories><repository>
  <id>culvert</id>
  <url>
    http://mavian.org/culvert/squeeze/main/
  </url>
</repository></repositories>
```

Figure 4: To use Culvert build script maintainers must configure Maven to use the Culvert repository server via the `<repositories>` section of their `pom.xml` file.

Package Name. Debian maintains its own naming policies by which each package is assigned a unique name within the Debian universe. For example, Java libraries tend to be named according to the pattern *lib*[name]-*java*, although some, such as `ant` and `junit` do not follow this pattern. We expose *package names* to Maven via the `<groupId>` element:

<groupid> `debian.libcommons-lang3-java` </groupId>

After the project developer makes their initial choices with respect to *edition*, *copyright-regime*, and of course the specific artifacts they need, Debian then decides which version of each library to download into the project based on these choices. Since Debian continuously revises their dependency graph, the specific version downloaded into the consuming project can change at any moment. The chances of a version update occurring in a given moment depend on the chosen dependency graph edition: *unstable* changes the most frequently (e.g., a few changes every day), and *old-stable* changes the least frequently with changes occurring even as rarely as once a year.

3.2 Using Culvert

To use Culvert, Java developers must modify their existing Maven Project Object Model `pom.xml` files in two ways. First, they must let Maven know about Culvert's URL. We are essentially employing the *Adapter* design pattern [11], but through HTTP instead of through an Object-Oriented paradigm. Because we use *Adapter*, Culvert appears and behaves like any compatible Maven repository of dependencies, and thus the simple XML addition shown (see Figure 4) allows a project's build script to connect to Culvert.

Next, as with any standard Maven build script, developers must identify the libraries they wish to integrate into their project, along with corresponding version information for each library. This information is specified using the `<dependencies>` section inside the project's build script. Users of Culvert must specify dependencies slightly differently compared to how they would when designing a traditional Maven build script. There are two differences:

1. Libraries in Culvert use different identifiers. While the final Jar name is usually identical, the `<groupId>` value for each library (a kind of namespace declaration in Maven) is different. In Culvert the `<groupId>` encodes and preserves information from the Debian package management system. For example, consider the `<artifactId>commons-lang3</artifactId>` artifact. The standard `<groupId>` for this artifact as found in `repo.maven.org` is:

   `org.apache.commons.commons-lang3`

   The translated `<groupId>` when using Culvert is:

   `debian.libcommons-lang3-java`

2. Culvert *mandates* the specification of version-ranges for each library. For most dependencies developers can specify the universal range, e.g., `<version>[0,)</version>` which usually resolves to "any version." If a developer so requires, they can also specify narrower ranges, e.g., `<version>[5.0.0, 6.0.0)</version>` which means "any 5.x.x version," and `<version>[5.0.0,)</version>` which means "any version 5.x.x or higher." Traditional Maven build scripts can also employ version ranges, but such is rare, and in fact, actively discouraged by many Java developers.[1]

Maven `<groupId>`	**Culvert `<groupID>`**
antlr	debian.libantlr-java
asm	debian.libasm3-java
bouncycastle	debian.libbcprov-java
org.apache.ant	debian.ant
org.apache.mina	debian.libmina2-java
org.codehaus.plexus	debian.libplexus-classworlds2-java
org.codehaus.plexus	debian.libplexus-containers1.5-java
org.codehaus.plexus	debian.libplexus-containers-java

Table 1: Here is a small sample of <groupId>'s as found in `repo.maven.org` compared to those found in Culvert for the same library dependency.

When developing Culvert we could have chosen identical `<groupId>` names as found in the Maven Central Repository (`repo.maven.org`). Such a design would allow build maintainers to more easily switch to Culvert: the migration would only require adding Culvert to the list of `<repositories>`. According to Maven's documentation, the Maven build tool would first check user-supplied repositories (e.g., Culvert) for a given dependency. Only when the initial lookup failed would Maven fall back to the default, **repo.maven.org**. Unfortunately, as we discovered during our case study, there are situations where developers need access to libraries from the original `repo.maven.org` central repository. Were we to use identical `<groupId>` names, artifacts in Culvert could potentially block/override access to related artifacts in Maven's default repository.

The naming scheme we chose provides a direct 1-for-1 mapping to Debian names. This pushes some work to a project's build script maintainer: for any given artifact, should they wish to use Culvert, they must identify the appropriate Culvert name for the library they wish to integrate into their system. However, in the long run this approach makes sense for our tool, for two reasons. First, when necessary, build scripts can contain a hybrid collection of dependencies, including some from Culvert, and others from `repo.maven.org`, since the unique identifiers should never collide. Second, we aim to be a transparent translation of Debian Java packages into a format that can be consumed by Maven. We have no interest in maintaining a Debian to `repo.maven.org` reverse-name-lookup database, since such would require ongoing work, and `repo.maven.org` naming policies are somewhat haphazard and continue to evolve. By anchoring Culvert on Debian's naming, and simply exposing Debian names in a consistent fashion for build maintainers to consume, there is no additional work required on our end.

[1] http://stackoverflow.com/questions/30571/

Figure 5: Culvert also offers a search mechanism. Users can search for Jars, and search results either match against Jar filenames or Debian's package descriptions.

4. RESULTS AND CONTRIBUTION

The system we studied, an online web-banking application used by hundreds of Canadian financial insitutions, was first deployed to production in 1999. The system had accumulated over 80 Java libraries during its 12 year evolution. including some components from an earlier dialup system deployed in 1994. We extracted 81 open-source components from this system, and in related work we determined authoritative provenance information for each of these components [12]. We chose this system for our case-study since it is highly active, moderately complex with many library dependencies, and it uses Maven for its build system.

For each of these dependencies simulated a "what if they switched to Culvert" scenario. In other words, we located the appropriate Debian package (if one existed) that contained the same dependency, and we compared the version numbers. We also analyzed the web-banking system's build scripts to see if any version-ranges were specified; the results of the analysis confirmed that all dependencies were pinned to explicit versions, supporting our initial hunch that Java developers do not generally use Maven's version-range feature.

The detailed results of our case study for each dependency are presented in our replication package.[2]. To evaluate reverse-compatibility of versions we assume all libraries are using the Eclipse/Apache 3-point version numbering scheme of *[major]:[minor]:[micro]* According to this scheme only libraries that differ in the first digit, the *major* digit, are said to be reverse-incompatible. We summarize our findings as following:

- Once a library appeared in an edition of Debian, it always remained (either the same version or newer) in later editions. This is not to say that Debian does not remove packages from their compilation, but we observed no removals among the 53 Debian packages identified in our case study.
- We chose the *Wheezy* edition of Debian to validate our case study. Conversely, we decided against validating with *Squeeze* or *Lenny*, since these are stable and proven production environments, and we imagined developers would prefer newer dependencies, as they themselves are developing cutting edge software.
- 53 of the 81 libraries (65%) could be found in Debian packages in one or more editions of the Debian distribution. After we configured Culvert to use *Wheezy*, Culvert chose 50 compatible library versions (62%) of which 9 were identical, and 41 were newer. Culvert also chose 5 reverse-incompatible newer library versions (6%).
- A final 28 libraries (35%) libraries were not present in any edition of Debian. Of these, 16 (20%) libraries represented functionality now included standard in the core Java libraries (e.g., in the JRE), and thus it makes sense Debian would not bother packaging these.
- Finally, we note that the security problem identified in our previous study was avoided in our simulation: Culvert chose a version of *spring.jar* that did not contain any known security holes.
- The potential legal problem we identified was not avoided. Debian does not contain *vreports.jar*, and thus the web banking system would likely continue to use this dependency from a different source (e.g., culvert would be unable to override this dependency). We also note that Debian does include recent versions of *itext.jar*, the underlying cause of the legal risk, and thus the online banking system could be vulnerable if they switched from *vreports.jar* to *itext.jar*. This problem arises because the GNU Affero license employed by *itext.jar* is compatible with Debian's policies, but is incompatible with the web banking system's policies.

4.1 Contribution

Culvert, the tool itself, represents the main contribution of this work. The majority of developers continue to pin their dependencies. Is this just a tradition, because there were no better ways? We hope Culvert can help us further investigate and understand developer needs around component management.

[2] `http://juliusdavies.ca/2012/acm-grand-finals/`

5. REFERENCES

[1] D. Coupland, *Microserfs*. Harper Perennial, 1996.

[2] M. M. Lehman, "Feedback in the software evolution process," *Information and Software Technology*, vol. 38, no. 11, pp. 681–686, November 1996.

[3] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 709–724, 2007.

[4] F. P. B. Jr., "No silver bullet - essence and accidents of software engineering," *IEEE Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[5] S. Fraser, F. P. B. Jr., M. Fowler, R. Lopez, A. Namioka, L. M. Northrop, D. L. Parnas, and D. A. Thomas, ""no silver bullet" reloaded: retrospective on "essence and accidents of software engineering"," in *OOPSLA Companion*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 1026–1030.

[6] J. J. Amor, G. Robles, J. M. González-Barahona, and F. Rivas, "Measuring Lenny: the size of Debian 5.0," Downloaded 2010-12-10, available at http://gsyc.es/~frivas/paper.pdf, 2009.

[7] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Trans. Software Eng.*, vol. 32, no. 12, pp. 952–970, 2006.

[8] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. ACM, 2007, pp. 204–213.

[9] S. P. Reiss, "Semantics-based code search," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 243–253.

[10] J. Davies, "Measuring subversions: security and legal risk in reused software artifacts," in *ICSE*, R. N. Taylor, H. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 1149–1151.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[12] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle, "Software bertillonage," *Empirical Software Engineering*, 2013 (to appear).

[13] S. McIntosh, B. Adams, and A. Hassan, "The evolution of java build systems," *Empirical Software Engineering*, pp. 1–31, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9169-5